

```

/*
 * SnapPea.h
 *
 * This file defines the interface between SnapPea's computational kernel
 * ("the kernel") and the user-interface ("the UI"). Both parts
 * must #include this file, and anything shared between the two parts
 * must be declared in this file. The only communication between the
 * two parts is via function calls -- no external variables are shared.
 *
 * All external symbols in the UI must begin with 'u' followed by a
 * capital letter. Nothing in the kernel should begin in this way.
 *
 * Typedef names use capitals for the first letter of each word,
 * e.g. Triangulation, CuspIndex.
 *
 * SnapPea 2.0 was funded by the University of Minnesota's
 * Geometry Center and the U.S. National Science Foundation.
 * SnapPea 3.0 is funded by the U.S. National Science Foundation
 * and the MacArthur Foundation. SnapPea and its source code may
 * be used freely for all noncommercial purposes. Please direct
 * questions, problems and suggestions to Jeff Weeks
 * (www.geometrygames.org/contact.html).
 *
 * Copyright 1999 by Jeff Weeks. All rights reserved.
 */

#ifndef _SnapPea_
#define _SnapPea_

/*
 * Note: values of the SolutionType enum are stored as integers in
 * the triangulation.doc file format. Changing the order of the
 * entries in the enum would therefore invalidate all previously stored
 * triangulations.
 */

typedef int SolutionType;
enum
{
    not_attempted,          /* solution not attempted, or user cancelled          ✓
    */
    geometric_solution,      /* all positively oriented tetrahedra; not flat or degenerate ✓
    */
    nongeometric_solution,   /* positive volume, but some negatively oriented tetrahedra ✓
    */
    flat_solution,          /* all tetrahedra flat, but no shapes = {0, 1, infinity} ✓
    */
    degenerate_solution,     /* at least one tetrahedron has shape = {0, 1, infinity} ✓
    */
    other_solution,         /* volume <= 0, but not flat or degenerate ✓
    */
    no_solution              /* gluing equations could not be solved ✓
    */
};

typedef int FuncResult;
enum
{
    func_OK = 0,
    func_cancelled,
    func_failed,
    func_bad_input
};

typedef struct
{
    double    real,
             imag;
} Complex;

typedef unsigned char    Boolean;

/*
 * The values of MatrixParity should not be changed.

```

```

* (They must correspond to the values in the parity[] table in tables.c.)
*/

typedef int MatrixParity;
enum
{
    orientation_reversing = 0,
    orientation_preserving = 1
};

/*
* SnapPea represents a Moebius transformation as a matrix
* in SL(2,C) plus a specification of whether the Moebius
* transformation is orientation_preserving or orientation_reversing.
*
* If mt->parity is orientation_preserving, then mt->matrix is
* interpreted in the usual way as the Moebius transformation
*
*          az + b
* f(z) = -----
*          cz + d
*
* If mt->parity is orientation_reversing, then mt->matrix is
* interpreted as a function of the complex conjugate z' ("z-bar")
*
*          az' + b
* f(z) = -----
*          cz' + d
*/

typedef Complex SL2CMatrix[2][2];

typedef struct
{
    SL2CMatrix    matrix;
    MatrixParity  parity;
} MoebiusTransformation;

/*
* Matrices in O(3,1) represent isometries in the Minkowski space
* model of hyperbolic 3-space. The matrices are expressed relative
* to a coordinate system in which the metric is
*
*          -1  0  0  0
*          0  1  0  0
*          0  0  1  0
*          0  0  0  1
*
* That is, the first coordinate is timelike, and the remaining
* three are spacelike. O(3,1) matrices represent both
* orientation_preserving and orientation_reversing isometries.
*/

typedef double O31Matrix[4][4];
typedef double GL4RMatrix[4][4];

/*
* An O31Vector is a vector in (3,1)-dimensional Minkowski space.
* The 0-th coordinate is the timelike one.
*/

typedef double O31Vector[4];

/*
* MatrixInt22 is a 2 x 2 integer matrix. A MatrixInt22
* may, for example, describe how the peripheral curves of
* one Cusp map to those of another.
*/

typedef int MatrixInt22[2][2];

/*
* An AbelianGroup is represented as a sequence of torsion coefficients.

```

```

* A torsion coefficient of 0 represents an infinite cyclic factor.
* For example, the group  $\mathbb{Z} + \mathbb{Z} + \mathbb{Z}/2 + \mathbb{Z}/5$  is represented as the
* sequence (0, 0, 2, 5). We make the convention that torsion coefficients
* are always nonnegative.
*
* The UI may declare pointers to AbelianGroups, but only the kernel
* may allocate or deallocate the actual memory used to store an
* AbelianGroup. (This allows the kernel to keep track of memory
* allocation/deallocation as a debugging aid.)
*/

typedef struct
{
    int          num_torsion_coefficients; /* number of torsion coefficients
    */
    long int     *torsion_coefficients;   /* pointer to array of torsion coefficients
    */
} AbelianGroup;

/*
* A closed geodesic may be topologically a circle or a mirrored interval.
*/

typedef int Orbifold1;
enum
{
    orbifold1_unknown,
    orbifold_s1,      /* circle */
    orbifold_mI       /* mirrored interval */
};

/*
* The following 2-orbifolds may occur as the link of an
* edge midpoint in a cell decomposition of a 3-orbifold.
*
* 94/10/4. The UI will see only types orbifold_nn
* and orbifold_xnn. Edges of the other types have 0-cells
* of the singular set at their midpoints, and are now
* subdivided in Dirichlet_extras.c. JRW
*/

typedef int Orbifold2;
enum
{
    orbifold_nn,      /* (nn) 2-sphere with two cone points (n may be 1) */
    orbifold_no,      /* (n|o) cross surface with cone point (n may be 1) */
    orbifold_xnn,     /* (*nn) disk with mirror boundary with two
    */
    /* corner reflectors */
    orbifold_2xn,     /* (2*n) disk with order two cone point and mirror
    */
    /* boundary with one corner reflector */
    orbifold_22n      /* (22n) sphere with three cone points */
};

/*
* A MultiLength records the complex length of a geodesic together with a
* parity telling whether it preserves or reverses orientation, a topology
* telling whether it's a circle or a mirrored interval, and a multiplicity
* telling how many distinct geodesics have that complex length, parity and
* topology.
*/

typedef struct
{
    Complex      length;
    MatrixParity parity;
    Orbifold1    topology;
    int          multiplicity;
} MultiLength;

/*
* A CuspNbhdHoroball records a horoball to be drawn as part of a
* picture of a cusp cross section. Only the kernel should allocate
* and free CuspNbhdHoroballs and CuspNbhdHoroballLists. These

```

```

* definitions are provided to the UI so it access the data easily.
*/

typedef struct
{
    Complex center;
    double radius;
    int cusp_index;
} CuspNbhdHoroball;

typedef struct
{
    /*
     * The horoball field points to an array
     * of num_horoballs CuspNbhdHoroballs.
     */
    int num_horoballs;
    CuspNbhdHoroball *horoball;
} CuspNbhdHoroballList;

/*
 * A CuspNbhdSegment records a 1-cell to be drawn as part of a
 * picture of a cusp cross section. (Typically it's either part of
 * a triangulation of the cusp cross section, or part of a Ford domain.)
 * Only the kernel should allocate and free CuspNbhdSegments and
 * CuspNbhdSegmentLists. These definitions are provided to the UI
 * so it can easily access the data.
 *
 * JRW 99/03/17 When the CuspNbhdSegment describes a triangulation
 * (as opposed to a Ford domain),
 *
 * the start_index tells the edge index of the vertical edge
 * that runs from the given segment's beginning
 * to the viewer's eye,
 *
 * the middle_index tells the edge index of the given segment, and
 *
 * the end_index tells the edge index of the vertical edge
 * that runs from the given segment's end
 * to the viewer's eye.
 *
 * These indices let the viewer see how the horoball picture
 * "connects up" to form the manifold.
 */

typedef struct
{
    Complex endpoint[2];
    int start_index,
        middle_index,
        end_index;
} CuspNbhdSegment;

typedef struct
{
    /*
     * segment is a pointer to an array of num_segments CuspNbhdSegments.
     */
    int num_segments;
    CuspNbhdSegment *segment;
} CuspNbhdSegmentList;

typedef int Orientability;
enum
{
    oriented_manifold,
    nonorientable_manifold,
    unknown_orientability
};

typedef int CuspTopology;

```

```

enum
{
    torus_cusp,
    Klein_cusp,
    unknown_topology
};

typedef int DirichletInteractivity;
enum
{
    Dirichlet_interactive,
    Dirichlet_stop_here,
    Dirichlet_keep_going
};

/*
 * An LRFactorization specifies the monodromy for a punctured torus
 * bundle over a circle. The factorization is_available whenever
 * (det(monodromy) = +1 and |trace(monodromy)| >= 2) or
 * (det(monodromy) = -1 and |trace(monodromy)| >= 1).
 * LR_factors points to an array of L's and R's, interpreted as factors
 *
 *          L = ( 1  0 )          R = ( 1  1 )
 *              ( 1  1 )          ( 0  1 )
 *
 * The factors act on a column vector, beginning with the last
 * (i.e. rightmost) factor.
 *
 * If negative_determinant is TRUE, the product is left-multiplied by
 *
 *          ( 0  1 )
 *          ( 1  0 )
 *
 * If negative_trace is TRUE, the product is left-multiplied by
 *
 *          (-1  0 )
 *          ( 0 -1 )
 *
 * When the factorization is unavailable, is_available is set to FALSE,
 * num_LR_factors is set to zero, and LR_factors is set to NULL.
 * But the negative_determinant and negative_trace flags are still set,
 * so the UI can display this information correctly.
 */
typedef struct
{
    Boolean is_available,
           negative_determinant,
           negative_trace;
    int    num_LR_factors;
    char   *LR_factors;
} LRFactorization;

/*
 * The full definition of a Shingling appears near the top of shingling.c.
 * But computationally a Shingling is just a collection of planes in
 * hyperbolic space (typically viewed as circles on the sphere at infinity).
 * Each plane has an index (which defines the color of the circle at
 * infinity).
 */

typedef struct
{
    /*
     * A plane in hyperbolic 3-space defines a hyperplane through
     * the origin in the Minkowski space model. Use the hyperplane's
     * normal vector to represent the original plane. [Note: the
     * normal is computed once, in the standard coordinate system,
     * and does not change as the UI rotates the polyhedron.]
     */
    O31Vector normal;

```

```

/*
 * A plane in hyperbolic 3-space intersects the sphere at infinity
 * in a circle. It's easy to draw the circle if we know its center
 * and two orthogonal "radials". (The 0-components of the center
 * and radials may be ignored.) [Note: the center and radials are
 * rotated in real time according to the polyhedron's current
 * position, and are scaled according to the window's pixel size.]
 */
O31Vector    center,
              radialA,
              radialB;

/*
 * The face planes of the original Dirichlet domain have index 0,
 * the face planes of the next layer (cf. shingling.c) have index 1,
 * and so on.
 */
int          index;
} Shingle;

typedef struct
{
    /*
     * A Shingling is just an array of Shingles.
     */
    int        num_shingles;
    Shingle    *shingles;
} Shingling;

/*
 * The following are "opaque typedefs". They let the UI declare and
 * pass pointers to Triangulations, IsometryLists, etc. without
 * knowing what a Triangulation, IsometryList, etc. is. The definitions
 * of struct Triangulation, struct IsometryList, etc. are private to the
 * kernel. SymmetryLists and IsometryLists are represented by the same
 * data structure because Symmetries are just Isometries from a manifold
 * to itself.
 */

typedef struct Triangulation      Triangulation;
typedef struct IsometryList      IsometryList;
typedef struct SymmetryGroup     SymmetryGroup;
typedef struct SymmetryGroupPresentation SymmetryGroupPresentation;
typedef struct DualOneSkeletonCurve DualOneSkeletonCurve;
typedef struct TerseTriangulation TerseTriangulation;
typedef struct GroupPresentation GroupPresentation;
typedef struct CuspNeighborhoods CuspNeighborhoods;
typedef struct NormalSurfaceList NormalSurfaceList;

/*
 * winged_edge.h describes the winged edge data structure used
 * to describe Dirichlet domains.
 */
#include "winged_edge.h"

/*
 * tersest_triangulation.h describes the most compressed form
 * for a Triangulation. The UI must have the actual definition
 * (not just an opaque typedef) because to read one from the
 * middle of a file it needs to know how long they are.
 */
#include "tersest_triangulation.h"

/*
 * link_projection.h describes the format in which the UI passes
 * link projections to the kernel.
 */
#include "link_projection.h"

```

```

* When the UI reads a Triangulation from disk, it passes the results
* to the kernel using the format described in triangulation_io.h.
*/
#include "triangulation_io.h"

/*
* covers.h defines a representation of a manifold's fundamental group
* into the symmetric group on n letters.
*/
#include "covers.h"

/* To guarantee thread-safety, it's useful to declare */
/* global variables to be "const", for example */
/* */
/* static const Complex minus_i = {0.0, -1.0}; */
/* */
/* Unfortunately the current gcc compiler complains when */
/* non-const variables are passed to functions expecting */
/* const arguments. Obviously this is harmless, but gcc */
/* complains anyhow. So for now let's use the following */
/* CONST macro, to allow the const declarations to be */
/* reactivated if desired. */
/* */
/* Note: In Win32, windef.h also defines CONST = const, */
/* so clear its definition before making our own. */
#undef CONST
#define CONST
/* #define CONST const */

#ifdef __cplusplus
extern "C" {
#endif

/*****/

/*
* The UI provides the following functions for use by the kernel:
*/

extern void uAcknowledge(const char *message);
/*
* Presents the string *message to the user and waits for acknowledgment ("OK").
*/

extern int uQuery(const char *message, const int num_responses,
                 const char *responses[], const int default_response);
/*
* Presents the string *message to the user and asks the user to choose
* one of the responses. Returns the number of the chosen response
* (numbering starts at 0). In an interactive context, the UI should
* present the possible responses evenhandedly -- none should be
* presented as a default. However, in a batch context (when no human
* is present), uQuery should return the default_response.
*/

extern void uFatalError(char *function, char *file);
/*
* Informs the user that a fatal error has occurred in the given
* function and file, and then exits.
*/

extern void uAbortMemoryFull(void);
/*
* Informs the user that the available memory has been exhausted,
* and aborts SnapPea.
*/

extern void uPrepareMemFullMessage(void);
/*
* uMemoryFull() is a tricky function, because the system may not find
* enough memory to display an error message. (I tried having it stash
* away some memory and then free it to support the desired dialog box,
* but at least on the Mac this didn't work for some unknown reason.)
*/

```

```

* uPrepareMemFullMessage() gives the system a chance to prepare
* a (hidden) dialog box. Call it once when the UI initializes.
*/

extern void          uLongComputationBegins(char *message, Boolean is_abortable);
extern FuncResult    uLongComputationContinues(void);
extern void          uLongComputationEnds(void);
/*
* The kernel uses these three functions to inform the UI of a long
* computation. The UI relays this information to the user in whatever
* manner it considers appropriate. For example, it might wait a second
* or two after the beginning of a long computation, and then display
* a dialog box containing *message (a typical message might be
* "finding canonical triangulation" or "computing hyperbolic structure").
* If is_abortable is TRUE, the dialog box would contain an abort button.
* The reason for waiting a second or two before displaying the dialog
* box is to avoid annoying the user with flashing dialog boxes for
* computations which turn out not to be so long after all.
*
* The kernel is responsible for calling uLongComputationContinues() at
* least every 1/60 second or so during a long computation.
* uLongComputationContinues() serves two purposes:
*
* (1) It lets the UI yield time to its window system. (This is
*      crucial for smooth background operation in the Mac's
*      cooperative multitasking environment. I don't know whether
*      it is necessary in X or NeXT.)
*
* (2) If the computation is abortable, it checks whether the user
*      has asked to abort, and returns the result (func_cancelled
*      to abort, func_OK to continue).
*
* While the kernel is responsible for making sure uLongComputationContinues()
* is called often enough, uLongComputationContinues() itself must take
* responsibility for not repeating time-consuming operations too often.
* For example, it might return immediately from a call if less than
* 1/60 of a second has elapsed since the last time it carried out
* its full duties.
*
* uLongComputationEnds() signals that the long computation is over.
* The kernel must call uLongComputationEnds() even after an aborted
* computation. ( uLongComputationContinues() merely informs the kernel
* that the user punched the abort button. The kernel must still call
* uLongComputationEnds() to dismiss the dialog box in the usual way.)
*
* If the UI receives a call to uLongComputationEnds() when no long
* computation is in progress, or a call to uLongComputationBegins()
* when a long computation is already in progress, it should notify
* the user of the error and exit.
*
* If the UI receives a call to uLongComputationContinues() when in
* fact no long computation is in progress, it should simply take
* care of any background responsibilities (see (1) above) and not
* complain. The reason for this provision is that the calls to
* uLongComputationBegins() and uLongComputationEnds() occur in high
* level functions, while the calls to uLongComputationContinues()
* occur at the lowest level, perhaps in a different file. Someday
* those low-level functions (for example, the routines for solving
* simultaneous linear equations) might be called as part of some quick,
* non-abortable computation.
*/

/*****
/*****

/*
* The kernel provides the following functions for use by the UI.
*
* A brief specification follows each function prototype.
* Complete documentation appears in the corresponding source file.
*/

```



```

/*****
*/
/*                      abelian_group.c                      */
/*                      */
/*****

extern void expand_abelian_group(AbelianGroup *g);
/*
 * Expands an AbelianGroup into its most factored form,
 * e.g.  $Z/2 + Z/2 + Z/4 + Z/3 + Z/9 + Z$ .
 * Each nonzero torsion coefficient is a power of a prime.
 */

extern void compress_abelian_group(AbelianGroup *g);
/*
 * Compresses an AbelianGroup into its least factored form,
 *  $Z/2 + Z/6 + Z/36 + Z$ .
 * Each torsion coefficient divides all subsequent torsion coefficients.
 */

extern void free_abelian_group(AbelianGroup *g);
/*
 * Frees the storage used to hold the AbelianGroup *g.
 */

/*****
*/
/*                      canonize.c                      */
/*                      canonize_part_1.c                  */
/*                      canonize_part_2.c                  */
/*                      */
/*****

extern FuncResult  canonize(Triangulation *manifold);

/*
 * Replaces the given Triangulation with the canonical retriangulation
 * of the canonical cell decomposition. Returns func_OK upon success,
 * func_failed if it cannot find a hyperbolic structure for *manifold.
 */

extern FuncResult  proto_canonize(Triangulation *manifold);
extern void        canonical_retriangulation(Triangulation *manifold);
/*
 * These functions comprise the two halves of canonize() in canonize.c.
 *
 * proto_canonize() replaces a Triangulation by the canonical
 * triangulation of the same manifold (if the canonical cell
 * decomposition is a triangulation) or by an arbitrary subdivision
 * of the canonical cell decomposition into Tetrahedra (if the canonical
 * cell decomposition contains cells other than tetrahedra).
 * Returns func_OK upon success, func_failed if it cannot find a
 * hyperbolic structure for *manifold.
 *
 * canonical_retriangulation() replaces the given subdivision of the
 * canonical cell decomposition with the canonical retriangulation.
 * This operation introduces finite vertices whenever the canonical cell
 * decomposition is not a triangulation to begin with. The hyperbolic
 * structure is discarded.
 */

extern Boolean     is_canonical_triangulation(Triangulation *manifold);
/*
 * Given a subdivision of the canonical cell decomposition as produced
 * by proto_canonize(), says whether it is the canonical decomposition
 * itself. In other words, it says whether the canonical cell decomposition
 * is a triangulation.
 */

/*****
*/

```

```

/*          change_peripheral_curves.c          */
/*          */
/*****/

extern FuncResult change_peripheral_curves(
    Triangulation *manifold,
    CONST MatrixInt22  change_matrices[]);
/*
 * If all the change_matrices have determinant +1, installs the
 * corresponding new peripheral curves and returns func_OK.
 * (See change_peripheral_curves.c for details.)
 * Otherwise does nothing and returns func_bad_input.
 */

/*****/
/*          chern_simons.c          */
/*          */
/*****/

extern void set_CS_value(    Triangulation  *manifold,
                           double          a_value);
/*
 * Set the Chern-Simons invariant of *manifold to a_value.
 */

extern void get_CS_value(    Triangulation  *manifold,
                           Boolean         *value_is_known,
                           double          *the_value,
                           int             *the_precision,
                           Boolean         *requires_initialization);
/*
 * If the Chern-Simons invariant of *manifold is known, sets
 * *value_is_known to TRUE and writes the current value and its precision
 * (the number of significant digits to the right of the decimal point)
 * to *the_value and *the_precision, respectively.
 *
 * If the Chern-Simons invariant is not known, sets *value_is_known to
 * FALSE, and then sets *requires_initialization to TRUE if the_value
 * is unknown because the computation has not been initialized, or
 * to FALSE if the_value is unknown because the solution contains
 * negatively oriented Tetrahedra. The UI might want to convey
 * these situations to the user in different ways.
 */

/*****/
/*          complex.c          */
/*          */
/*****/

extern Complex  complex_minus      (Complex z0, Complex z1),
               complex_plus       (Complex z0, Complex z1),
               complex_mult       (Complex z0, Complex z1),
               complex_div        (Complex z0, Complex z1),
               complex_sqrt       (Complex z),
               complex_conjugate   (Complex z),
               complex_negate     (Complex z),
               complex_real_mult  (double r, Complex z),
               complex_exp        (Complex z),
               complex_log        (Complex z, double approx_arg);
extern double   complex_modulus    (Complex z);
extern double   complex_modulus_squared (Complex z);
extern Boolean  complex_nonzero    (Complex z);
extern Boolean  complex_infinite   (Complex z);
/*
 * The usual complex arithmetic functions.
 *
 * Standard complex constants (Zero, One, etc.) are defined in the kernel.
 */

```

```

/*****
/*
/*                                     complex_length.c
/*
/*                                     */
/*****/

extern Complex complex_length_mt(MoebiusTransformation *mt);
extern Complex complex_length_o31(O31Matrix m);
/*
 * Computes the complex length of an isometry. Please see
 * complex_length.c for full definitions and explanations.
 * complex_length_mt() and complex_length_o31() are identical except
 * for the form in which the input is given.
 */

/*****
/*
/*                                     continued_fractions.c
/*
/*                                     */
/*****/

extern Boolean appears_rational(double x0, double x1, double confidence,
                                long *num, long *den);
/*
 * Checks whether a finite-precision real number x known to lie in the
 * interval (x0, x1) appears to be a rational number p/q. If it does,
 * it sets *num and *den to p and q, respectively, and returns TRUE.
 * Otherwise it sets *num and *den to 0 and returns FALSE.
 * The confidence parameter gives the maximal acceptable probability
 * of a "false positive".
 */

/*****
/*
/*                                     core_geodesic.c
/*
/*                                     */
/*****/

extern void core_geodesic( Triangulation *manifold,
                           int cusp_index,
                           int singularity_index,
                           Complex core_length,
                           int precision);
/*
 * Examines the Cusp of index cusp_index in *manifold.
 *
 * If the Cusp is unfilled or the Dehn filling coefficients are not
 * integers, sets *singularity_index to zero and leaves *core_length
 * undefined.
 *
 * If the Cusp has relatively prime integer Dehn filling coefficients,
 * sets *singularity_index to 1 and *core_length to the complex length
 * of the central geodesic.
 *
 * If the Cusp has non relatively prime integer Dehn filling coefficients,
 * sets *singularity_index to the index of the singular locus, and
 * *core_length to the complex length of the central geodesic in the
 * smallest manifold cover of a neighborhood of the singular set.
 *
 * In the latter two cases, if the precision pointer is not NULL,
 * *precision is set to the number of decimal places of accuracy in
 * the computed value of *core_length.
 *
 * core_geodesic() is intended for use by the UI. Kernel function may
 * find compute_core_geodesic() (declared in kernel_prototypes.h) more
 * convenient.
 */

/*****
/*
/*                                     cover.c
/*
/*                                     */
/*****/

```

```

/*
/*****

Triangulation *construct_cover( Triangulation      *base_manifold,
                               RepresentationIntoSn *representation,
                               int                  n);

/*
 * Constructs the n-sheeted cover of the given base_manifold defined
 * by the given transitive representation.
 */

/*****
/*
/*          current_curve_basis.c
/*
/*****

extern void current_curve_basis(      Triangulation *manifold,
                                     int            cusp_index,
                                     MatrixInt22    basis_change);

extern void install_current_curve_bases(Triangulation *manifold);

/*
 * current_curve_basis() accepts a Triangulation and a cusp index,
 * and computes a 2 x 2 integer matrix basis_change with the property
 * that
 *
 *     if the Cusp of index cusp_index is filled, and has
 *         relatively prime integer Dehn filling coefficients,
 *
 *         the first row of basis_change is set to the current
 *         Dehn filling coefficients, and
 *         the second row of basis_change is set to the shortest
 *         curve which completes a basis.
 *
 *     else
 *
 *         basis_change is set to the identity
 *
 * install_current_curve_bases() installs the above basis
 * on all cusps of the manifold.
 */

/*****
/*
/*          cusp_neighborhoods.c
/*
/*****

extern CuspNeighborhoods *initialize_cusp_neighborhoods(
                               Triangulation *manifold);

/*
 * Initializes a CuspNeighborhoods data structure.
 * It works with a copy of manifold, leaving the original untouched.
 * It does all indicated Dehn fillings.
 * Returns a pointer to the CuspNeighborhoods structure upon success,
 * of NULL if the "manifold" isn't a cusped hyperbolic 3-manifold.
 */

extern void free_cusp_neighborhoods(
                               CuspNeighborhoods *cusp_neighborhoods);

/*
 * Frees the CuspNeighborhoods structure, including the copy of
 * the Triangulation it contains.
 */

extern int get_num_cusp_neighborhoods(
                               CuspNeighborhoods *cusp_neighborhoods);

/*
 * Returns the number of cusps. This will be the number of unfilled
 * cusps in the original manifold, which may be less than the total

```

```

*   number of cusps.
*/

extern CuspTopology get_cusp_neighborhood_topology(
                                CuspNeighborhoods *cusp_neighborhoods,
                                int cusp_index);

/*
*   Returns the CuspTopology of the given cusp.
*/

extern double get_cusp_neighborhood_displacement(
                                CuspNeighborhoods *cusp_neighborhoods,
                                int cusp_index);

/*
*   Returns the (linear) displacement of the horospherical cross
*   section of the given cusp from its home position. At the home
*   position the cusp cross section has area  $(3/8)\sqrt{3}$  and
*   encloses a volume of  $(3/16)\sqrt{3}$  in the cusp. At its home
*   position, a cusp cannot overlap itself, nor can it overlap any
*   other cusp which does not already overlap itself. Please see
*   cusp_neighborhoods.c for details.
*/

extern Boolean get_cusp_neighborhood_tie(
                                CuspNeighborhoods *cusp_neighborhoods,
                                int cusp_index);

/*
*   Says whether this cusp's neighborhood is tied to other cusps'.
*/

extern double get_cusp_neighborhood_cusp_volume(
                                CuspNeighborhoods *cusp_neighborhoods,
                                int cusp_index);

/*
*   Returns the volume enclosed by the horospherical cross section
*   of the given cusp.
*/

extern double get_cusp_neighborhood_manifold_volume(
                                CuspNeighborhoods *cusp_neighborhoods);

/*
*   Returns the volume of the manifold.
*/

extern Triangulation *get_cusp_neighborhood_manifold(
                                CuspNeighborhoods *cusp_neighborhoods);

/*
*   Returns a pointer to a copy of the manifold. The UI may do as it
*   pleases with the copy, and should free it when it's done.
*/

extern double get_cusp_neighborhood_reach(
                                CuspNeighborhoods *cusp_neighborhoods,
                                int cusp_index);

/*
*   Returns the displacement at which the cusp cross section first
*   bumps into itself.
*/

extern double get_cusp_neighborhood_max_reach(
                                CuspNeighborhoods *cusp_neighborhoods);

/*
*   Returns the maximum reach over the whole manifold.
*/

extern double get_cusp_neighborhood_stopping_displacement(
                                CuspNeighborhoods *cusp_neighborhoods,
                                int cusp_index);

extern int get_cusp_neighborhood_stopper_cusp_index(
                                CuspNeighborhoods *cusp_neighborhoods,
                                int cusp_index);

/*
*   Return the displacement at which the cusp first bumps into another
*   cusp (or possibly into itself), and the cusp it bumps into.
*/

```

```

* Unlike the reach, the stopper and the stopping displacement depend
* on the current displacements of all the cusps in the triangulation.
* They vary dynamically as the user moves the cusp cross sections.
*/

extern void set_cusp_neighborhood_displacement(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index,
    double new_displacement);

/*
* Sets the cusp neighborhood's displacement to the requested value,
* clipping it to the range [0, stopping_displacement] if necessary.
* Recomputes the canonical cell decomposition.
*/

extern void set_cusp_neighborhood_tie(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index,
    Boolean new_tie);

/*
* Tells the kernel whether this cusp's neighborhood should be
* tied to other cusps (which have previously been "tied").
* The kernel makes all tied cusps have the same displacement.
*/

extern void get_cusp_neighborhood_translations(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index,
    Complex *meridian,
    Complex *longitude);

/*
* Returns the meridional and longitudinal translation vectors
* for the given cusp cross section, taking into account its current
* displacement. For a Klein bottle cusp, the longitudinal translation
* will be that of the double cover. As a convenience, the longitude
* will always point in the x-direction.
*/

extern CuspNbhdHoroballList *get_cusp_neighborhood_horoballs(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index,
    Boolean full_list,
    double cutoff_height);

/*
* Returns a list of horoballs seen from the given cusp, taking into
* account the cusp cross sections' current displacements. Only one
* translate is given for each horoball -- to draw the full picture the
* UI must find all visible translates using the meridian and longitude
* provided by get_cusp_neighborhood_translations(). For a Klein bottle
* cusp, get_cusp_neighborhood_horoballs() reports data for the double
* cover. If full_list is TRUE, get_cusp_neighborhood_horoballs()
* reports all horoballs whose Euclidean height in the upper half space
* model is at least cutoff_height. If full_list is FALSE, it reports
* only a few of the largest horoballs (the cutoff_height is ignored).
* This lets the UI draw a simpler picture while the user is changing
* something in real time, and then draw a more complete picture afterwards.
*/

extern void free_cusp_neighborhood_horoball_list(
    CuspNbhdHoroballList *horoball_list);

/*
* Frees a CuspNbhdHoroballList when the UI's done with it.
*/

extern CuspNbhdSegmentList *get_cusp_neighborhood_triangulation(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index);

/*
* Returns a list of edges in the restriction of the canonical cell
* decomposition to the cusp cross section, taking into account the
* cusp cross section's current displacement. Only one translate is
* given for each edge -- to draw the full picture the UI must find all
* visible translates using the meridian and longitude provided by
* get_cusp_neighborhood_translations(). For a Klein bottle cusp,

```

```

/*
 * get_cusp_neighborhood_triangulation() reports data for the double cover.
 */

extern CuspNbhdSegmentList *get_cusp_neighborhood_Ford_domain(
    CuspNeighborhoods *cusp_neighborhoods,
    int cusp_index);

/*
 * Returns a list of edges in the Ford domain, taking into account the
 * cusp cross section's current displacement. Only one translate is
 * given for each edge -- to draw the full picture the UI must find all
 * visible translates using the meridian and longitude provided by
 * get_cusp_neighborhood_translations(). For a Klein bottle cusp,
 * get_cusp_neighborhood_Ford_domain() reports data for the double cover.
 */

extern void free_cusp_neighborhood_segment_list(
    CuspNbhdSegmentList *segment_list);

/*
 * Frees a CuspNbhdSegmentList when the UI's done with it.
 */

/*****
 *
 * Dirichlet.c
 *
 *****/

extern WEPolyhedron *Dirichlet(
    Triangulation *manifold,
    double vertex_epsilon,
    Boolean centroid_at_origin,
    DirichletInteractivity interactivity,
    Boolean maximize_injectivity_radius);

/*
 * Computes a Dirichlet domain for the given manifold or orbifold.
 * Returns NULL if the Dehn filling coefficients are not all integers,
 * of if roundoff errors lead to topological problems.
 * Returns a pointer to the Dirichlet domain otherwise.
 */

extern WEPolyhedron *Dirichlet_with_displacement(
    Triangulation *manifold,
    double displacement[3],
    double vertex_epsilon,
    Boolean centroid_at_origin,
    DirichletInteractivity interactivity,
    Boolean maximize_injectivity_radius);

/*
 * Like Dirichlet(), only allows an arbitrary displacement
 * of the basepoint. The displacement is in tangent space
 * coordinates, so the distances can't be interpreted too literally.
 * Reasonable displacements are to the order of 0.1.
 * Large displacements are possible, but degrade the numerical
 * accuracy of the resulting Dirichlet domain.
 */

extern WEPolyhedron *Dirichlet_from_generators(
    O3Matrix generators[],
    int num_generators,
    double vertex_epsilon,
    DirichletInteractivity interactivity,
    Boolean maximize_injectivity_radius);

/*
 * Like Dirichlet(), only starts with a set of O(3,1) matrix generators
 * instead of a Triangulation.
 */

extern WEPolyhedron *Dirichlet_from_generators_with_displacement(
    O3Matrix generators[],
    int num_generators,
    double displacement[3],
    double vertex_epsilon,
    DirichletInteractivity interactivity,
    Boolean maximize_injectivity_radius);

```

[illegible]



```

extern Triangulation *Dirichlet_to_triangulation(WEPolyhedron *polyhedron);
/*
 * Converts a Dirichlet domain to a Triangulation, leaving the
 * Dirichlet domain unchanged. For closed manifolds, drills out
 * an arbitrary curve and expresses the manifold as a Dehn filling.
 */

/*****
/*
double_cover.c
/*
*****/

extern Triangulation *double_cover(Triangulation *manifold);
/*
 * Returns a pointer to the double cover of the nonorientable
 * Triangulation *manifold.
 */

/*****
/*
dual_curves.c
/*
*****/

extern void dual_curves(    Triangulation    *manifold,
                           int              max_size,
                           int              *num_curves,
                           DualOneSkeletonCurve ***the_curves);
/*
 * Computes a reasonable selection of simple closed curves in
 * a manifold's dual 1-skeleton.
 */

extern void get_dual_curve_info(    DualOneSkeletonCurve *the_curve,
                                    Complex                *complete_length,
                                    Complex                *filled_length,
                                    MatrixParity            *parity);
/*
 * Reports the complex length of a curve in the dual 1-skeleton,
 * relative to both the complete and filled hyperbolic structures,
 * and also its parity (orientation_preserving or orientation_reversing).
 */

extern void free_dual_curves(    int              num_curves,
                                DualOneSkeletonCurve ***the_curves);
/*
 * Frees the array of curves computed by dual_curves().
 */

/*****
/*
drilling.c
/*
*****/

extern Triangulation *drill_cusp(    Triangulation    *old_manifold,
                                    DualOneSkeletonCurve *curve_to_drill,
                                    char                *new_name);
/*
 * Drills a curve out of the dual 1-skeleton of an n-cusp manifold to
 * create an (n+1)-cusp manifold.
 */

/*****
/*
filling.c
/*
*****/

```

```

extern Triangulation *fill_cusps(    Triangulation    *manifold,
                                     Boolean           fill_cusp[],
                                     char              *new_name,
                                     Boolean           fill_all_cusps);

/*
 * Permanently fills k of the cusps of an n-cusp manifold.
 * Typically fill_all_cusps is FALSE, and the function returns
 * an ideal Triangulation of the resulting (n - k)-cusp manifold.
 * fill_cusp[] is a Boolean array specifying which k cusps (k < n)
 * are to be filled.
 *
 * In the exceptional case that fill_all_cusps is TRUE, the function
 * returns a triangulation with finite vertices only.
 * Such triangulations are unacceptable for most SnapPea routines,
 * and should be used only for writing to disk. When fill_all_cusps
 * is TRUE, fill_cusp is ignored and may be NULL.
 *
 * new_name is the name to be given to the new Triangulation.
 */

extern Triangulation *fill_reasonable_cusps(Triangulation *manifold);
/*
 * Makes reasonable choices for fill_cusp[] and new_name, and calls
 * fill_cusps(). Specifically, it will fill all cusps with relatively
 * prime Dehn filling coefficients, unless this would leave no cusps
 * unfilled, in which case it leaves cusp 0 unfilled. It copies the
 * name from the original manifold.
 */

extern Boolean cusp_is_fillable(Triangulation *manifold, int cusp_index);
/*
 * Returns TRUE if a cusp has relatively prime integer Dehn filling
 * coefficients, FALSE otherwise.
 */

extern Boolean is_closed_manifold(Triangulation *manifold);
/*
 * Returns TRUE iff all cusps are filled and the coefficients
 * are relatively prime integers.
 */

/*****
 *
 *                               */
 *                               */
 *                               */
 *****/

extern GroupPresentation *fundamental_group(
    Triangulation    *manifold,
    Boolean           simplify_presentation,
    Boolean           fillings_may_affect_generators,
    Boolean           minimize_number_of_generators);
/*
 * Computes the fundamental group of the manifold, taking into account
 * Dehn fillings, and returns a pointer to it. Please see
 * fundamental_group.c for an explanation of the arguments.
 */

extern int fg_get_num_generators(GroupPresentation *group);
/*
 * Returns the number of generators in the GroupPresentation.
 */

extern Boolean fg_integer_fillings(GroupPresentation *group);
/*
 * Says whether the underlying space is a manifold or orbifold,
 * as opposed to some other generalized Dehn filling.
 */

extern FuncResult fg_word_to_matrix(
    GroupPresentation *group,
    int               *word,
    *group,
    *word,

```

```

        O3lMatrix          result_O3l,
        MoebiusTransformation *result_Moebius);

/*
 * Converts an abstract word in the fundamental group to a matrix
 * in the matrix representation. The abstract word is given as a
 * string of integers. The integer 1 means the first generator,
 * 2 means the second, etc., while -1 is the inverse of the first
 * generator, -2 is the inverse of the second, etc. The integer 0
 * indicates the end of the string. The result is given both as
 * an O3lMatrix and a MoebiusTransformation. Returns func_OK if
 * successful, or func_bad_input if the input word is not valid.
 */

extern int fg_get_num_relations(GroupPresentation *group);
/*
 * Returns the number of relations in the GroupPresentation.
 */

extern int *fg_get_relation( GroupPresentation *group,
                             int which_relation);
/*
 * Returns the specified relation (using 0-based indexing).
 * It allocates the memory for it, so you should pass the pointer
 * back to fg_free_relation() when you're done with it.
 * Each relation is a string of integers. The integer 1 means
 * the first generator, 2 means the second, etc., while -1 is the
 * inverse of the first generator, -2 is the inverse of the second, etc.
 * The integer 0 indicates the end of the string.
 */

extern void fg_free_relation(int *relation);
/*
 * Frees a relation allocated by fg_get_relation().
 */

extern int fg_get_num_cusps(GroupPresentation *group);
/*
 * Returns the number of cusps of the underlying manifold.
 * This *includes* the filled cusps. So, for example, if you do (5,1)
 * Dehn filling on the figure eight knot complement, you can see the
 * words in the fundamental group corresponding to the (former!) cusp's
 * meridian and longitude.
 */

extern int *fg_get_meridian( GroupPresentation *group,
                             int which_cusp);

extern int *fg_get_longitude( GroupPresentation *group,
                              int which_cusp);
/*
 * Returns the word corresponding to a meridian or longitude, in the
 * same format used by fg_get_relation() above. They allocate the
 * memory for the string of integers, so you should pass the pointer
 * back to fg_free_relation() when you're done with it. Meridians and
 * longitudes are available whether the cusps are filled or not, as
 * explained for fg_get_num_cusps() above.
 */

extern int *fg_get_original_generator( GroupPresentation *group,
                                       int which_generator);
/*
 * Returns a word which expresses one of the standard geometric
 * generators (as defined in choose_generators.c) in terms of the
 * group presentation's generators. The word is in the same format
 * used by fg_get_relation() above. Note that which_generator is
 * given relative to 0-based indexing, but the letters in the word
 * you get out use 1-based numbering, as in fg_get_relation().
 * Please free the word with fg_free_relation() when you're done.
 */

extern void free_group_presentation(GroupPresentation *group);
/*
 * Frees the storage occupied by a GroupPresentation.
 */

```

```

/*****
/*
/*          homology.c
/*
/*
/*****/

extern AbelianGroup *homology(Triangulation *manifold);
/*
*   If all Dehn filling coefficients are integers, returns a pointer to
*   the first homology group of *manifold. In particular, it will
*   happily compute homology groups of orbifolds. If one or more Dehn
*   filling coefficients are not integers, returns NULL. This function
*   allocates the memory for the AbelianGroup; the UI should call
*   free_abelian_group() (no pun intended) to release it.
*
*   96/12/11 Checks for overflows, and returns NULL if any occur.
*/

extern AbelianGroup *homology_from_fundamental_group(
    GroupPresentation *group);
/*
*   Abelianizes a group presentation and returns the result.
*   Returns NULL if overflows occur.
*/

/*****
/*
/*          hyperbolic_structure.c
/*
/*
/*****/

extern SolutionType find_complete_hyperbolic_structure(Triangulation *manifold);
/*
*   Attempts to find a complete hyperbolic structure for the
*   Triangulation *manifold. Sets the solution_type[complete] member of
*   *manifold to the type of solution found. If this type is anything
*   other than no_solution, stores the hyperbolic structure by setting
*   the *shape[complete] field of each Tetrahedron in the Triangulation. The
*   solution is also stored as the initial filled solution, by setting the
*   solution_type[filled] member of *manifold and the *shape[filled] fields
*   of the Tetrahedra; the is_complete flag of each Cusp is set to TRUE.
*
*   The hyperbolic structure is computed using Newton's method, beginning
*   with all tetrahedra regular.
*
*   Returns: the type of solution found.
*/

extern SolutionType do_Deahn_filling(Triangulation *manifold);
/*
*   Attempts to find a hyperbolic structure for a *manifold, based on
*   the current Dehn filling coefficients. Sets the solution_type[filled]
*   member of *manifold to the type of solution found. If
*   this type is anything other than no_solution, stores the hyperbolic
*   structure by setting the *shape[filled] field of each Tetrahedron in
*   the Triangulation.
*
*   The hyperbolic structure is computed using Newton's method; the
*   initial guess is the previous Dehn filled solution.
*
*   Returns: the type of solution found.
*/

extern SolutionType remove_Deahn_fillings(Triangulation *manifold);
/*
*   Removes all Dehn fillings.
*
*   Returns: the type of solution restored.
*/

```

```

/*****
/*
/*                                     index_to_hue.c
/*                                     */
/*****/

extern double index_to_hue(int index);
/*
 * Maps the nonnegative integers to a set of easily distinguishable hues.
 *
 * index    0        1        2        3        4        5        6        . . .
 * hue      0        1/2      1/4      3/4      1/8      5/8      3/8      . . .
 */

extern double horoball_hue(int index);
/*
 * Provides hand chosen hues for indices 0-5, and uses index_to_hue()
 * to interpolate thereafter. The hope is for nicer looking horoball
 * packings.
 */

/*****
/*
/*                                     interface.c
/*                                     */
/*****/

extern char *get_triangulation_name(Triangulation *manifold);
/*
 * Return a pointer to the name of the Triangulation *manifold.
 * The pointer points to the actual name, not a copy.
 */

extern void set_triangulation_name(Triangulation *manifold, char *new_name);
/*
 * Sets the Triangulation's name to new_name.
 */

extern SolutionType get_complete_solution_type(Triangulation *manifold);
/*
 * Returns the SolutionType of the complete structure.
 */

extern SolutionType get_filled_solution_type(Triangulation *manifold);
/*
 * Returns the SolutionType of the current Dehn filling.
 */

extern int get_num_tetrahedra(Triangulation *manifold);
/*
 * Returns the number of tetrahedra in the Triangulation *manifold.
 */

extern Orientability get_orientability(Triangulation *manifold);
/*
 * Returns the orientability of *manifold.
 */

extern int get_num_cusps(Triangulation *manifold);
/*
 * Returns the number of cusps in *manifold.
 */

extern int get_num_or_cusps(Triangulation *manifold);
/*
 * Returns the number of orientable cusps in *manifold.
 */

extern int get_num_nonor_cusps(Triangulation *manifold);
/*
 * Returns the number of nonorientable cusps in *manifold.
 */

```

```

extern int get_max_singularity(Triangulation *manifold);
/*
 * Returns the maximum value of gcd(m,l) over all integer Dehn filling
 * coefficients (m,l) for filled cusps in *manifold.
 */

extern int get_num_generators(Triangulation *manifold);
/*
 * Returns the number of generators being used to represent *manifold's
 * fundamental group.
 */

extern void get_cusp_info( Triangulation *manifold,
                          int cusp_index,
                          CuspTopology *topology,
                          Boolean *is_complete,
                          double *m,
                          double *l,
                          Complex *initial_shape,
                          Complex *current_shape,
                          int *initial_shape_precision,
                          int *current_shape_precision,
                          Complex *initial_modulus,
                          Complex *current_modulus);
/*
 * Provides information about the cusp whose index is cusp_index in
 * *manifold. (The cusp indices run from zero to one less than the
 * number of cusps.)
 *
 * *topology is set to torus_cusp, Klein_cusp, or unknown_topology.
 * *is_complete is set to TRUE if the cusp is not Dehn filled, and
 * FALSE if it is.
 * *m and *l are set to the current Dehn filling coefficients.
 * They will be meaningful only if the cusp is filled.
 * If the cusp is nonorientable, only *m will be meaningful
 * (because *l must be zero for a Klein bottle cusp -- see
 * the comment at the top of holonomy.c).
 * *initial_shape is set to the initial shape (longitude/meridian) of the
 * cusp, i.e. the shape it had when all cusps were unfilled.
 * *current_shape is set to the cusp's current shape if the cusp is_complete,
 * zero otherwise.
 * *initial_shape_precision is set to the number of decimal places of accuracy
 * in the computed value of initial_shape.
 * *current_shape_precision is set to the number of decimal places of accuracy
 * in the computed value of current_shape.
 * *initial_modulus is set to the modulus ( (second shortest translation)/
 * (shortest translation) ) of the initial cusp shape.
 * *current_modulus is set to the modulus of the current cusp shape.
 *
 * You may pass NULL for pointers to values you aren't interested in.
 */

extern FuncResult set_cusp_info(Triangulation *manifold,
                              int cusp_index,
                              Boolean *is_complete,
                              double m,
                              double l);
/*
 * Looks for a cusp with index cusp_index in Triangulation *manifold.
 * If not found,
 * alerts the user and exits (this should never occur
 * unless there is a bug in the UI).
 * If found,
 * if cusp_is_complete is TRUE,
 * sets the is_complete field of the cusp to TRUE, and
 * sets the Dehn filling coefficients to 0.0,
 * if cusp_is_complete is FALSE
 * sets the is_complete field of the cusp to FALSE, and
 * sets the Dehn filling coefficients to m and l.
 *
 * set_cusp_info() checks for errors in the values of m and l.
 * The (0,0) Dehn filling is never allowed, and only (p,0) fillings are
 * allowed on nonorientable cusps. If an error is detected, the cusp
 * will be left unchanged.

```

```

*
* Returns:
*     func_OK           for success
*     func_bad_input    for illegal Dehn filling coefficients
*/

extern void get_holonomy(    Triangulation    *manifold,
                             int              cusp_index,
                             Complex          *meridional_holonomy,
                             Complex          *longitudinal_holonomy,
                             int              *meridional_precision,
                             int              *longitudinal_precision);

/*
* Passes back the holonomies of the meridian and longitude,
* and an estimate of their precision (number of decimal
* digits to the right of the decimal point).
*/

extern void get_tet_shape(    Triangulation    *manifold,
                             int              which_tet,
                             Boolean          fixed_alignment,
                             double          *shape_rect_real,
                             double          *shape_rect_imag,
                             double          *shape_log_real,
                             double          *shape_log_imag,
                             int              *precision_rect_real,
                             int              *precision_rect_imag,
                             int              *precision_log_real,
                             int              *precision_log_imag,
                             Boolean          *is_geometric);

/*
* Provides information about the shape of the Tetrahedron in
* position which_tet in the linked list (which_tet takes a value
* in the range [0, (#tetrahedra - 1)] ). (Note: which_tet
* does not explicitly refer to the "index" field of the Tetrahedron
* data structure, although in practice it will coincide.)
* get_tet_shape() provides the shape of the Tetrahedron in both
* rectangular and logarithmic forms, relative to whatever coordinate
* system was used most recently. This means that the rectangular
* form will satisfy  $|z| < 1$  and  $|z - 1| < 1$ . The last four arguments
* give the precision of the preceding four, expressed as the number
* of significant decimal digits following the decimal point.
* (Warning: the precision is only a rough estimate. The last
* digit or two may sometimes be incorrect.) The flag *is_geometric
* is set to TRUE iff all dihedral angles lie in the range [0,pi].
*/

extern int get_num_edge_classes(    Triangulation    *manifold,
                                   int              edge_class_order,
                                   Boolean          greater_than_or_equal);

/*
* If greater_than_or_equal == TRUE, returns the number of EdgeClasses
* whose order is greater than or equal to edge_class_order.
* If greater_than_or_equal == FALSE, returns the number of EdgeClasses
* whose order is exactly edge_class_order.
*/

/*****
*/
/*
/*
/*
/*
/*****/

extern FuncResult compute_isometries(
                                   Triangulation    *manifold0,
                                   Triangulation    *manifold1,
                                   Boolean          *are_isometric,
                                   IsometryList     **isometry_list,
                                   IsometryList     **isometry_list_of_links);

/*
* Checks whether manifold0 and manifold1 are isometric (taking into

```

```

*   account the Dehn fillings).  If manifold0 and manifold1 are cusped
*   manifolds, sets *isometry_list and *isometry_list_of_links as
*   in compute_cusped_isometries() below.  Returns
*       func_OK           if all goes well,
*       func_bad_input    if some Dehn filling coefficients are not
*                           relatively prime integers,
*       func_failed       if it can't decide.
*/

extern int isometry_list_size(IsometryList *isometry_list);
/*
*   Returns the number of Isometries in the IsometryList.
*/

extern int isometry_list_num_cusps(IsometryList *isometry_list);
/*
*   Returns the number of cusps in each of the underlying manifolds.
*   If the IsometryList is empty (as would be the case when the
*   underlying manifolds have different numbers of cusps), then
*   isometry_list_num_cusps()'s return value is undefined.
*/

extern void isometry_list_cusp_action(   IsometryList    *isometry_list,
                                         int              anIsometryIndex,
                                         int              aCusp,
                                         int              *cusp_image,
                                         int              cusp_map[2][2]);
/*
*   Fills in the cusp_image and cusp_map[2][2] to describe the action
*   of the given Isometry on the given Cusp.
*/

extern Boolean isometry_extends_to_link(IsometryList *isometry_list, int i);
/*
*   Returns TRUE if Isometry i extends to the associated links (i.e. if it
*   takes meridians to meridians), FALSE if it doesn't.
*/

extern void isometry_list_orientations(
    IsometryList    *isometry_list,
    Boolean          *contains_orientation_preserving_isometries,
    Boolean          *contains_orientation_reversing_isometries);
/*
*   Says whether the IsometryList contains orientation-preserving
*   and/or orientation-reversing elements.  Assumes the underlying
*   Triangulations are oriented.
*/

extern void free_isometry_list(IsometryList *isometry_list);
/*
*   Frees the IsometryList.
*/

/*****
*/
/*
/*           isometry_cusped.c
/*
/*
/*****/

extern Boolean same_triangulation(   Triangulation    *manifold0,
                                     Triangulation    *manifold1);
/*
*   Check whether manifold0 and manifold1 have combinatorially
*   equivalent triangulations (ignoring Dehn fillings).
*   This function is less versatile than a call to
*   compute_isometries(manifold0, manifold1, &are_isometric, NULL, NULL)
*   but it's useful for batch processing, when you want to avoid the
*   overhead of constantly recomputing canonical retriangulations.
*/

/*****
*/

```



```

/*                      length_spectrum.c                      */
/*                                                                */
/*****

extern void length_spectrum(    WEPolyhedron    *polyhedron,
                              double           cutoff_length,
                              Boolean           full_rigor,
                              Boolean           multiplicities,
                              double           user_radius,
                              MultiLength      **spectrum,
                              int              *num_lengths);

/*
 * Takes as input a manifold in the form of a Dirichlet domain, and
 * finds all geodesics of length less than or equal to cutoff_length.
 * Please length_spectrum.c for details.
 */

extern void free_length_spectrum(MultiLength *spectrum);
/*
 * Deallocates the memory used to store the length spectrum.
 */

/*****
/*                                                                */
/*                      link_complement.c                      */
/*                                                                */
/*****

extern Triangulation *triangulate_link_complement(
                              KLPPProjection *aLinkProjection);
/*
 * Triangulate the complement of aLinkProjection.
 */

/*****
/*                                                                */
/*                      matrix_conversion.c                    */
/*                                                                */
/*****

extern void Moebius_to_O3l(MoebiusTransformation *A, O3lMatrix B);
extern void O3l_to_Moebius(O3lMatrix B, MoebiusTransformation *A);
/*
 * Convert matrices back and forth between SL(2,C) and O(3,1).
 */

extern void Moebius_array_to_O3l_array( MoebiusTransformation  arrayA[],
                                         O3lMatrix              arrayB[],
                                         int                     num_matrices);
extern void O3l_array_to_Moebius_array( O3lMatrix              arrayB[],
                                         MoebiusTransformation arrayA[],
                                         int                     num_matrices);
/*
 * Convert arrays of matrices back and forth between SL(2,C) and O(3,1).
 */

extern Boolean O3l_determinants_OK( O3lMatrix  arrayB[],
                                    int         num_matrices,
                                    double      epsilon);
/*
 * Returns TRUE if all the O3lMatrices in the array have determinants
 * within epsilon of plus or minus one, and FALSE otherwise.
 */

/*****
/*                                                                */
/*                      matrix_generators.c                   */
/*                                                                */
/*****

extern void matrix_generators( Triangulation      *manifold,

```

```

                                MoebiusTransformation generators[],
                                Boolean                centroid_at_origin);

/*
 * Computes the MoebiusTransformations representing the action
 * of the generators of a manifold's fundamental group on the sphere at
 * infinity. Writes the MoebiusTransformations to the array generators[],
 * which it assumes has already been allocated. You may use
 * get_num_generators() to determine how long an array to allocate.
 * If centroid_at_origin is TRUE, the initial tetrahedron is positioned
 * with its centroid at the origin; otherwise the initial tetrahedron
 * is positioned with its vertices at {0, 1, infinity, z}.
 */

/*****
/*
 *                               */
/*                               */
/*                               */
/*****

extern void verify_my_malloc_usage(void);
/*
 * The UI should call verify_my_malloc_usage() upon exit to verify that
 * the number of calls to my_malloc() was exactly balanced by the number
 * of calls to my_free(). In case of error, verify_my_malloc_usage()
 * passes an appropriate message to uAcknowledge.
 */

/*****
/*
 *                               */
/*                               */
/*                               */
/*****

extern FuncResult find_normal_surfaces( Triangulation      *manifold,
                                       NormalSurfaceList  **surface_list);

/*
 * Tries to find connected, embedded normal surfaces of nonnegative
 * Euler characteristic. If spheres or projective planes are found,
 * then tori and Klein bottles aren't reported, because from the point
 * of view of the Geometrization Conjecture, one wants to cut along
 * spheres and projective planes first. Surfaces are guaranteed to be
 * connected. They aren't guaranteed to be incompressible, although
 * typically they are. There is no guarantee that all such normal
 * surfaces will be found. Returns its result as a pointer to a
 * NormalSurfaceList, the internal structure of which is private to
 * the kernel. To get information about the normal surfaces on the list,
 * use the functions below. To split along a normal surface, call
 * split_along_normal_surface(). When you're done with the
 * NormalSurfaceList, free it using free_normal_surfaces().
 *
 * The present implementation works only for cusped manifolds.
 * Returns func_bad_input for closed manifolds, or non-manifolds.
 */

extern int      number_of_normal_surfaces_on_list(
                NormalSurfaceList  *surface_list);

/*
 * Returns the number of normal surfaces contained in the list.
 */

extern Boolean  normal_surface_is_orientable(
                NormalSurfaceList  *surface_list,
                int                 index);
extern Boolean  normal_surface_is_two_sided(
                NormalSurfaceList  *surface_list,
                int                 index);
extern int      normal_surface_Euler_characteristic(
                NormalSurfaceList  *surface_list,
                int                 index);

/*
 * Return information about a given normal surface on the list.
 * The indices run from 0 through (number of surfaces - 1).
 */

```

```

*/

extern void free_normal_surfaces(NormalSurfaceList *surface_list);
/*
 * Frees an array of NormalSurfaceLists.
 */

/*****
/*
/*          normal_surface_splitting.c
/*
/*
/*****

extern FuncResult split_along_normal_surface(
                                NormalSurfaceList *surface_list,
                                int index,
                                Triangulation *pieces[2]);

/*
 * Splits the manifold (stored privately in the NormalSurfaceList)
 * along the normal surface of the given index (indices range from 0 to
 * (number of surfaces - 1)). If the normal surface is a 2-sided
 * projective plane, split_along_normal_surface() returns func_bad_input;
 * otherwise it returns func_OK. If the normal surface is a sphere or
 * 1-sided projective plane, the resulting spherical boundary component(s)
 * are capped off with 3-ball(s); otherwise the new torus or Klein bottle
 * boundary component(s) become cusp(s). If the normal surface is
 * nonseparating, the result is returned in pieces[0], and pieces[1]
 * is set to NULL. If the normal surface is separating, the two pieces
 * are returned in pieces[0] and pieces[1].
 */

/*****
/*
/*          o3l_matrices.c
/*
/*
/*****

/*
 * Most of the functions in o3l_matrices.c are private to the kernel.
 * The following have been made available to the UI as well.
 */
extern double      gl4R_determinant(GL4RMatrix m);
extern double      o3l_trace(O3lMatrix m);

/*****
/*
/*          orient.c
/*
/*
/*****

extern void reorient(Triangulation *manifold);
/*
 * Reverse a manifold's orientation.
 */

/*****
/*
/*          punctured_torus_bundles.c
/*
/*
/*****

extern void bundle_LR_to_monodromy( LRFactorization *anLRFactorization,
                                MatrixInt22 aMonodromy);
/*
 * Multiplies out anLRFactorization to obtain aMonodromy.
 */

extern void bundle_monodromy_to_LR( MatrixInt22 aMonodromy,
                                LRFactorization **anLRFactorization);
/*

```

```

*   If (det(aMonodromy) = +1 and |trace(aMonodromy)| >= 2) or
*   (det(aMonodromy) = -1 and |trace(aMonodromy)| >= 1),
*   then bundle_monodromy_to_LR() conjugates aMonodromy to a
*   nonnegative or nonpositive matrix, and factors it as
*   anLRFactorization. These cases include all monodromies of
*   hyperbolic manifolds, as well as the nonhyperbolic cases
*   (det(aMonodromy) = +1 and |trace(aMonodromy)| = 2), which
*   the user might want to see factored just for fun.
*   Otherwise bundle_monodromy_to_LR() sets
*   (*anLRFactorization)->is_available to FALSE, but nevertheless
*   sets negative_determinant and negative_trace correctly in case
*   the UI wants to display them. The UI should indicate that the
*   factorization is not available (e.g. by displaying "N/A") so
*   the user doesn't confuse this case with an empty factorization.
*/

extern LRFactorization *alloc_LR_factorization(int aNumFactors);
extern void free_LR_factorization(LRFactorization *anLRFactorization);
/*
 *   Allocates/frees LRFactorizations.
 */

extern Triangulation *triangulate_punctured_torus_bundle(
    LRFactorization *anLRFactorization);
/*
 *   If the manifold is hyperbolic (i.e. if the number of LR factors
 *   is at least two for an orientable bundle, or at least one for a
 *   nonorientable bundle), triangulates the complement and returns
 *   a pointer to it. Otherwise returns NULL.
 */

/*****
/*
/*                               */
/*               rehydrate_census.c               */
/*                               */
*****/

extern void rehydrate_census_manifold(
    TersestTriangulation    tersest,
    int                     which_census,
    int                     which_manifold,
    Triangulation           **manifold);
/*
 *   Rehydrates a census manifold from a tersest description, resolving
 *   any ambiguities in the choice of peripheral curves for the cusps.
 */

/*****
/*
/*                               */
/*               representations.c               */
/*                               */
*****/

RepresentationList *find_representations(    Triangulation    *manifold,
    int                                     n,
    PermutationSubgroup range);
/*
 *   Finds all transitive representations of a manifold's fundamental
 *   group into Z/n or S(n), for use in constructing n-sheeted covers.
 *   To dispose of the RepresentationList when you're done, use
 *   free_representation_list() below.
 */

void free_representation_list(
    RepresentationList *representation_list);
/*
 *   Frees a RepresentationList.
 */

/*****
/*
*****/

```

```

/*                                     shingling.c                                     */
/*                                     */
/*****

extern Shingling *make_shingling(WEPolyhedron *polyhedron, int num_layers);
/*
 * Constructs the shingling defined by the given Dirichlet domain.
 * Please see the top of shingling.c for detailed documentation.
 */

extern void free_shingling(Shingling *shingling);
/*
 * Releases the memory occupied by the shingling.
 */

extern void compute_center_and_radials( Shingle      *shingle,
                                       O3lMatrix    position,
                                       double        scale);
/*
 * Uses shingle->normal along with the given position and scale to
 * compute shingle->center, shingle->radialA and shingle->radialB.
 */

/*****
/*                                     */
/*                                     shortest_cusp_basis.c                                     */
/*                                     */
/*****

extern Complex cusp_modulus(Complex cusp_shape);
/*
 * Accepts a cusp_shape (longitude/meridian) and returns the cusp modulus.
 * Loosely speaking, the cusp modulus is defined as
 * (second shortest translation)/(shortest translation); it is a complex
 * number z lying in the region  $|\operatorname{Re}(z)| \leq 1/2$  &&  $|z| \geq 1$ . If z lies
 * on the boundary of this region, we choose it so that  $\operatorname{Re}(z) \geq 0$ .
 */

extern void shortest_cusp_basis(      Complex      cusp_shape,
                                   MatrixInt22 basis_change);
/*
 * Accepts a cusp_shape (longitude/meridian) and computes the 2 x 2 integer
 * matrix which transforms the old basis (u, v) = (meridian, longitude)
 * to the new basis (u', v') = (shortest, second shortest).
 */

extern Complex transformed_cusp_shape(      Complex      cusp_shape,
                                           CONST MatrixInt22 basis_change);
/*
 * Accepts a cusp_shape and a basis_change, and computes the shape of the
 * cusp relative to the transformed basis. The transformed basis may or
 * may not be the (shortest, second shortest) basis.
 */

extern void install_shortest_bases( Triangulation *manifold);
/*
 * Installs the (shortest, second shortest) basis on each torus Cusp
 * of manifold, but does not change the bases on Klein bottle cusps.
 */

/*****
/*                                     */
/*                                     simplify_triangulation.c                                     */
/*                                     */
/*****

extern void basic_simplification(Triangulation *manifold);
/*
 * Simplifies the triangulation in a speedy yet effective manner.
 */

extern void randomize_triangulation(Triangulation *manifold);

```

```

/*
 * Randomizes the Triangulation, and then resimplifies it.
 */

/*****
/*
 *                               sl2c_matrices.c
 *                               */
 *                               */
 *****/

/*
 * Most of the functions in sl2c_matrices.c are private to the kernel.
 * The following has been made available to the UI as well.
 */

extern Complex sl2c_determinant(CONST SL2CMatrix m);
/*
 * Returns the determinant of m.
 */

/*****
/*
 *                               symmetry_group.c
 *                               */
 *                               */
 *****/

extern FuncResult compute_symmetry_group(
    Triangulation    *manifold,
    SymmetryGroup    **symmetry_group_of_manifold,
    SymmetryGroup    **symmetry_group_of_link,
    Triangulation    **symmetric_triangulation,
    Boolean          *is_full_group);
/*
 * Computes the SymmetryGroup of a closed or cusped manifold.
 * If the manifold is cusped, also computes the SymmetryGroup of the
 * corresponding link (defined at the top of symmetry_group_cusped.c).
 */

extern void free_symmetry_group(SymmetryGroup *symmetry_group);
/*
 * Frees a SymmetryGroup.
 */

/*****
/*
 *                               symmetry_group_info.c
 *                               */
 *                               */
 *****/

extern Boolean symmetry_group_is_abelian(    SymmetryGroup    *symmetry_group,
                                           AbelianGroup      **abelian_description);
/*
 * If the SymmetryGroup is abelian, sets *abelian_description to point
 * to the SymmetryGroup's description as an AbelianGroup, and returns TRUE.
 * Otherwise sets *abelian_description to NULL and returns FALSE.
 */

extern Boolean symmetry_group_is_dihedral(SymmetryGroup *symmetry_group);
/*
 * Returns TRUE if the SymmetryGroup is dihedral, FALSE otherwise.
 */

extern Boolean symmetry_group_is_polyhedral(SymmetryGroup *symmetry_group,
                                           Boolean        *is_full_group,
                                           int            *p,
                                           int            *q,
                                           int            *r);
/*
 * Returns TRUE if the SymmetryGroup is polyhedral, FALSE otherwise.
 * If the SymmetryGroup is polyhedral, reports whether it's the full group
 * (binary polyhedral, not just plain polyhedral), and reports the values

```

```

*   for (p,q,r). The pointers for is_full_group, p, q and r may be NULL
*   if this information is not desired.
*/

extern Boolean symmetry_group_is_S5(SymmetryGroup *symmetry_group);
/*
*   Returns TRUE if the SymmetryGroup is the symmetric group on 5 letters,
*   FALSE otherwise.
*/

extern Boolean symmetry_group_is_direct_product(SymmetryGroup *symmetry_group);
/*
*   Returns TRUE if the SymmetryGroup is a nontrivial, nonabelian direct
*   product, FALSE otherwise.
*/

extern SymmetryGroup *get_symmetry_group_factor(SymmetryGroup *symmetry_group,
                                                int factor_number);
/*
*   If the SymmetryGroup is a nontrivial, nonabelian direct product,
*   returns a pointer to factor "factor_number" (factor_number = 0 or 1).
*   Otherwise returns NULL. This is a pointer to the internal data
*   structure -- not a copy! -- so please don't free it.
*/

extern Boolean symmetry_group_is_amphicheiral(SymmetryGroup *symmetry_group);
/*
*   Returns TRUE if the SymmetryGroup contains orientation-reversing
*   elements, FALSE otherwise. Assumes the underlying manifold is oriented.
*/

extern Boolean symmetry_group_invertible_knot(SymmetryGroup *symmetry_group);
/*
*   Assumes the underlying manifold is oriented and has exactly
*   one Cusp. Returns TRUE if some Symmetry acts on the Cusp
*   via the matrix  $\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$ ; returns FALSE otherwise.
*/

extern int symmetry_group_order(SymmetryGroup *symmetry_group);
/*
*   Returns the order of the SymmetryGroup.
*/

extern int symmetry_group_product(SymmetryGroup *symmetry_group, int i, int j);
/*
*   Returns the product of group elements i and j. We use the
*   convention that products of symmetries read right to left.
*   That is, the composition  $\text{symmetry}[i] \circ \text{symmetry}[j]$  acts by
*   first doing  $\text{symmetry}[j]$ , then  $\text{symmetry}[i]$ .
*/

extern int symmetry_group_order_of_element(SymmetryGroup *symmetry_group, int i);
/*
*   Returns the order of group element i.
*/

extern IsometryList *get_symmetry_list(SymmetryGroup *symmetry_group);
/*
*   Returns the list of "raw" Isometries comprising a SymmetryGroup.
*/

extern SymmetryGroup *get_commutator_subgroup(SymmetryGroup *symmetry_group);
extern SymmetryGroup *get_abelianization(SymmetryGroup *symmetry_group);
/*
*   Compute the commutator subgroup  $[G,G]$  and the abelianization  $G/[G,G]$ .
*   The UI should eventually use free_symmetry_group() to free them.
*/

extern SymmetryGroup *get_center(SymmetryGroup *symmetry_group);
/*
*   Computes the center of G, which is the subgroup consisting of
*   elements which commute with all elements in G.
*   The UI should eventually use free_symmetry_group() to free it.
*/

```

```

extern SymmetryGroupPresentation *get_symmetry_group_presentation(
    SymmetryGroup *symmetry_group);
/*
 * Returns a presentation for the given SymmetryGroup.
 * The internal structure of the SymmetryGroupPresentation is private
 * to the kernel; use the functions below to get information about it.
 * When you're done with it, use free_symmetry_group_presentation()
 * to free the storage.
 */

extern int sg_get_num_generators(SymmetryGroupPresentation *group);
/*
 * Returns the number of generators in the SymmetryGroupPresentation.
 */

extern int sg_get_num_relations(SymmetryGroupPresentation *group);
/*
 * Returns the number of relations in the SymmetryGroupPresentation.
 */

extern int sg_get_num_factors( SymmetryGroupPresentation *group,
    int which_relation);
/*
 * Returns the number of factors in the specified relation.
 * For example, the relation  $a^3 * b^{-2} * c^5$  has three factors.
 * The parameter which_relation uses 0-based indexing.
 */

extern void sg_get_factor( SymmetryGroupPresentation *group,
    int which_relation,
    int which_factor,
    int *generator,
    int *power);
/*
 * Reports the generator and power of the specified factor in the
 * specified relation. For example, if relation 1 (i.e. the second
 * relation) is  $a^3 * b^{-2} * c^5$ , then passing which_relation = 1 and
 * which_factor = 2 will cause it to report *generator = 2 and
 * *power = 5.
 */

extern void free_symmetry_group_presentation(SymmetryGroupPresentation *group);
/*
 * Frees the storage occupied by a SymmetryGroupPresentation.
 */

/*****
 *
 * terse_triangulation.c
 *
 *****/

extern TerseTriangulation *tri_to_terse(Triangulation *manifold);
extern TerseTriangulation *tri_to_canonical_terse(
    Triangulation *manifold,
    Boolean respect_orientation);
/*
 * tri_to_terse() accepts a pointer to a Triangulation, computes
 * a corresponding TerseTriangulation, and returns a pointer to it.
 * tri_to_canonical_terse() is similar, but chooses the
 * TerseTriangulation which is "least" among all possible choices
 * of base Tetrahedron and base Permutation.
 */

extern Triangulation *terse_to_tri(TerseTriangulation *tt);
/*
 * Accepts a pointer to a TerseTriangulation, expands it to a full
 * Triangulation, and returns a pointer to it.
 */

extern void free_terse_triangulation(TerseTriangulation *tt);
/*

```



```

* Releases the memory used to store a TerseTriangulation.
*/

/*****
/*
/*          tersest_triangulation.c          */
/*
/*          ****
/*****

extern void terse_to_tersest(    TerseTriangulation    *terse,
                                TersestTriangulation    tersest);

/*
* Converts a TerseTriangulation to a TersestTriangulation.
*/

extern void tersest_to_terse(    TersestTriangulation    tersest,
                                TerseTriangulation    **terse);

/*
* Converts a TersestTriangulation to a TerseTriangulation.
* Allocates space for the result.
*/

extern void tri_to_tersest(      Triangulation          *manifold,
                                TersestTriangulation    tersest);

/*
* Composes tri_to_terse() and terse_to_tersest().
*/

extern void tersest_to_tri(      TersestTriangulation    tersest,
                                Triangulation          **manifold);

/*
* Composes tersest_to_terse() and terse_to_tri().
*/

/*****
/*
/*          triangulations.c          */
/*
/*          ****
/*****

extern void data_to_triangulation( TriangulationData    *data,
                                  Triangulation          **manifold_ptr);

/*
* Uses the TriangulationData (defined in triangulation_io.h) to
* construct a Triangulation. Sets *manifold_ptr to point to the
* Triangulation, or to NULL if it fails.
*/

extern void triangulation_to_data( Triangulation          *manifold,
                                  TriangulationData    **data_ptr);

/*
* Allocates the TriangulationData and writes in the data describing
* the manifold. Sets *data_ptr to point to the result. The UI
* should call free_triangulation_data() when it's done with the
* TriangulationData.
*/

extern void free_triangulation_data(TriangulationData *data);

/*
* If the UI lets the kernel allocate a TriangulationData structure
* (as in a call to triangulation_to_data()), then the UI should
* call free_triangulation_data() to release it.
* If the UI allocates its own TriangulationData structure (as in
* preparing for a call to data_to_triangulation()), then the UI
* should release the structure itself.
*/

extern void free_triangulation(Triangulation *manifold);

/*
* If manifold != NULL, frees up the storage associated with a
* triangulation structure.
* If manifold == NULL, does nothing.

```

```

*/

extern void copy_triangulation(Triangulation *source, Triangulation **destination);
/*
 * Makes a copy of the Triangulation *source.
 */

/*****
/*
/*                               */
/*               two_bridge.c      */
/*                               */
/*                               */
*****/

extern void two_bridge( Triangulation *manifold,
                       Boolean *is_two_bridge, long int *p, long int *q);
/*
 * Checks whether *manifold is the (conjectured) canonical triangulation
 * of a 2-bridge knot or link complement. If it is, sets *is_two_bridge
 * to TRUE and writes the fraction p/q describing the knot or link into
 * (*p)/(*q). If it's not, sets *is_two_bridge to FALSE and leaves *p
 * and *q undefined.
 */

/*****
/*
/*                               */
/*               volume.c          */
/*                               */
/*                               */
*****/

extern double volume(Triangulation *manifold, int *precision);
/*
 * Computes and returns the volume of the manifold.
 * If the pointer "precision" is not NULL, estimates the number
 * of decimal places of accuracy, and places the result in the
 * variable *precision.
 */

#ifdef __cplusplus
}
#endif

#endif

```